Perihelion Software

Technical Report No. 17

Upgrading from Helios 1.0 to 1.1

by
Andy Evans

May 1989

Perihelion Software Limited
The Maltings
Charlton Road
Shepton Mallet
Somerset
BA4 5QE
England
Telephone +44 749 4203
Fax. +44 749 4977

Copyright (c) 1989 Perihelion Software Ltd.

Permission to copy this technical note without fee is hereby granted, provided that the copyright message and this permission appears in all copies.

Introduction

Helios 1.1 contains a number of improvements and bug fixes. The kernel has been completely rewritten in C for ease of maintenance and for portability. Despite this, the kernel is faster than the previous one.

Helios 1.1 is upwards compatible with V1.0. Any program which runs under 1.0 will run under 1.1; although, programs compiled under 1.1 will NOT run properly under 1.0. You are strongly advised to recompile your programs to take advantage of the improvements in the compiler.

The network server and task force manager have also been rewritten. Note that the format of resource maps has changed slightly; full details are given later.

The way in which Helios starts up is different and full details of the file, initrc, are provided later in this document; users who are upgrading from version 1.0 should note that there is no requirement to launch system servers such as the network server from the file loginrc.

Starting Helios

When Helios starts, it executes init, which in turn reads the text file, etc/initrc. By default, this text file contains a command line which instructs init to execute login. login checks the directory, etc, for a file called passwd, and if it is present it requests the user to login. As Helios 1.1 is distributed with an example password file, you will need to supply a user name in response to the login prompt. This name must be one of those listed in the password file, so you could enter the following:

```
tim
```

When you have successfully logged into Helios, you can then edit the password file to include your own name. Helios will then allow you to enter your name in response to the login prompt.

Resource Maps

The etc subdirectory contains suitable resource maps for simple transputer configurations. Note that the format of resource maps has changed between V1.0 and this version of Helios. You must use the new version of rmgen to construct the maps, and maps will need a specification of the master node and the name of the device driver used to reset nodes. The driver im_ra_b4.d is used for B004/B008 style boards. The driver pa_ra.d is used for Parsytec boards; in this case you must also copy the file lib/pa_rboot on top of the normal lib/rboot.

The system is supplied with a resource map called etc/default.map which is the binary version of the text resource map etc/default.rm. This defines the smallest possible configuration consisting of a single processor connected to the I/O server.

Once you are happy with Helios and have read the chapter in the manual concerning resource maps and networks then you will want to create a new resource map which reflects the layout of your own transputer network. When you have done this you should compile the text map with rmgen and place the result in etc/default.map, or alter the initrc file which contains a reference to this filename. Then reboot Helios and you should get a log of the nodes which have been booted. Error messages will be given if the physical configuration of the network does not match that specified in the map. *You should note that the 1988 edition of the manual describes the resource map for Helios 1.0. The reader is advised to consult the 1989 edition (published by Prentice Hall) or the appendix to this technical note for the syntax used by Helios 1.1.*

You can run tasks in remote processors by using the Component Distribution Language or by using the command, remote. The CDL compiler, described later in this note, allows you to automatically distribute tasks among the available processors. remote is used to execute the specified task on a specified processor; this command is fully described in *The Helios Operating System*.

I/O Server

The server now includes a device called */logger* that can be used as the destination for debug information. Any text sent to this device is diverted to a file, the server window, or both file and window. The default destination for information sent to this device is the screen, but this can be changed by including the entry "logging_destination" within the host.con file. Alternatively, the current destination can be changed at any time by typing <ctrl><shift>L; this character sequence instructs the server to toggle between each of the three available options. By default, all information that is diverted to a file is sent to /helios/logfile. A different filename can be specified by applying the logfile entry within the host.con file.

The new logger device is used by all programs which call the system function, IOdebug(). This means that all debugging messages generated by Helios are sent to /logger, as are those which are generated by the user's own programs. The server will also send its own debugging information to the logger.

In addition to /logger, the new server defines an action for the keyboard sequence, <ALT>F2. Previous versions of the server allowed the user to use <ALT>F1 to move forwards through its list of virtual windows. The latest server allows the new key sequence to be used to move backwards through this list.

The Configuration File

Some additional options have been defined for use with the host.con file.

logging_destination
: This option is used to specify the destination for information which is sent to /logger. By default, the destination is the screen, but can be changed by applying this option with one of three qualifiers: screen, file or both. If the information is to be diverted to a file, it will be sent to /helios/logfile, unless an alternative filename is specified using the *logfile* option

logfile
: All information which is diverted to a file by /logger is sent to the file, /helios/logfile, by default. This can be changed by specifying a filename as a qualifier for this option.

server_windows
: The PC server supports multiple pseudo-windows. Each window takes up the entire screen, but the server remembers the current state for each window and updates the screen when you switch between windows. Output to windows continues even when the window is not displayed. This windowing system can be disabled by omitting the *server_windows* entry from the configuration file; this is useful when another window manager, such as X-Windows, is being supplied on the transputer side.

server_window_nopop
: The server has its own window which is used to display error messages and debugging information. By default, this window moves to the top of the window stack when text is written to it, but this action can be disabled by including the *server_window_nopop* option within the host.con file.

rs232_interrupt
: If you are planning on using the serial ports for any special devices, such as a mouse, then you will need to include this option in your host.con file. By default, the server takes over both interrupt vectors for the serial ports, but if you attach a device such as a mouse, then you will have to instruct the server to use just one of the interrupt vectors; this will leave the other available for the device driver. If you wished to use just one communications port, com2, the following two entries should appear in the host.con file

```
rs232_ports = 2
rs232_interrupt = 2
```

rs232_ports

The PC server provides complete control over the communications ports, but does so in a way that may cause problems if used with non-standard hardware. This option allows the user to specify which serial ports are to be used.

```
rs232_ports = 1,2
```

The above example would instruct the server to use com1 and com2. If this entry is absent the server will not make use of the serial ports.

Multiple Windows

The server supports its own windowing system which is enabled by including the *server_windows* option in the host.con file. Helios 1.1 is shipped with an alternative windowing system which is provided by the file, lib/window. To use this windowing system remove the *server_windows* option from the host.con file, and reboot Helios. Note, however, that the server's own windowing system provides a better environment that that provided by lib/window, and that the latter is provided for compatibility with Helios 1.0.

Startup Options

When Helios 1.1 is booted into a transputer that is connected to the I/O processor, it executes /helios/lib/init, which in turn reads the text file, etc/initrc. This text file can be edited by the user to define the way in which Helios is to start, and can contain any of the commands listed below.

 # comment

Any line in initrc which starts with '#' is treated as a comment and is ignored.

auto *servername*

This command adds the specified name into the name server. This means that the server will be loaded on demand from /helios/lib when it is first used.

console *servername windowname ...*

This command creates one or more windows using the specified server, and provides the window as output for commands that are executed from within initrc. If this is omitted any output from tasks such as the network server will go to the /logger device. Note that the standard initrc file contains an entry at the end to run login; this will only work if a console command has been given in order to create a window. If no console has been created then the login process must be created by the batch server by adding a suitable entry in the file, *etc/batchrc*, which explicitly provides an environment for login.

ifabsent *servername command*

This command determines if the specified server exists, and if it does not it treats the rest of the line as a command which it will execute. *command* must be one of the commands listed in this section.

run *[-e] [-w] pathname [argv ...]*

This command is used to execute the command whose full pathname is passed as an argument. The -e option is used to pass the environment to the command, and the -w option is used to force *run* to wait for the command to terminate before it terminates itself. The following argument vector, argv, consists of the command name and each of its arguments; it is only used if the -e option is specified. Any program started using run -e will subsequently have standard streams to a window created by *console*.

waitfor *server_name*

This command waits for the specified server to be loaded before it terminates. If the server has not been loaded, the waitfor command will result in it being loaded (assuming that the name is defined).

The following is a listing of the configuration file which is shipped with Helios 1.1.

```
# Helios System Configuration File
# This file is interpreted by init to configure the system
# it is NOT a shell script.
ifabsent /window auto /window
waitfor /window
console /window console
run -e /helios/bin/startns startns -r /helios/etc/default.map
waitfor /tfm
#run -e /helios/lib/sm sm
run -e /helios/lib/bs bs
#run -e /helios/bin/smlogin -
run -e /helios/bin/login -
```

The first two command lines in this file are used to install the window manager. This is achieved by adding the server's name into the name server by use of **auto**, and then waiting for the server to be loaded by **waitfor**. This may appear to be a somewhat verbose way of loading the window server, but it is necessary because **run** alone would only load the server; it would not create an entry within the name server. It should be noted, however, that some server's add their name into the name server themselves, and can therefore be loaded successfully by **run**.

Having created the window server, the program uses the **console** command to create a window called "console". The environment provided by this window is then noted by **init**, so that it can be passed on to subsequent **run** commands that are invoked with the **-e** option. In this configuration file, all subsequent **run** commands are passed this environment so that their output is sent to the console window instead of /logger.

After applying the **run** command to **startns**, the batch server is loaded. One of the first tasks that the batch server performs is to read the job description file, etc/batchrc. In this release of Helios the job description file is empty, but it can be modified by the user. Full details of the syntax are given in the description of the batch server found elsewhere in these notes.

Job Control in the Shell

When a command is executed in the background using the shell metacharacter, &, it is referred to as a job. Whenever the shell creates a job it assigns it a job number and a process identification number (process id). It then displays these identifiers to the user in a line of the form:

```
[<job_number>] <process_id>
```

A complete list of all the current jobs and their associated numbers are maintained by the shell, and can be viewed by use of the **jobs** command.

When a job has been created there are two operations which the user can perform on it; it can be brought to the foreground or it can be terminated. For either operation it will be necessary to know the jobs name.

Job Name	Description
%<*job_number*>	The job with the specified job number.
%%	The current job.
%+	The current job.
%-	The previous job.

Each of the character sequences shown above can be supplied as an argument to **kill** or **fg** to terminate the specified job or bring it to the foreground. The character sequences can also be entered directly into the shell, and will have the same effect as if they were supplied as arguments to **fg**. Full details of **kill** and **fg** are given later in this note. See *New Commands in Helios 1.1*.

Whenever a job terminates, the shell removes the job from its internal list and displays a message of the form:

```
[<job_number>] <exit_status> <command>
```

The exit_status in this message takes one of three forms. If the command terminated successfully the message is "Done"; if the command exited with a non-zero exit code, then the message takes the form "Exit *n*", where *n* is the the argument which was passed to the exit function. If the command was terminated by a signal, then the message describes that signal. For example, if job number 1 was terminated because of stack overflow, the shell displays:

```
[1]  Stack Overflow  myprog
```

Alias Server

A new server, lib/alias, has been provided which allows the user to assign an alias to a directory name. The syntax for this server is

/helios/lib/alias <name> <directory>

This command should always be run in the background, either by appending the metacharacter, &, when entering the command into the shell, or by using the run command in the startup file, initrc. A typical application of this command is:

```
/helios/lib/alias etc /helios/etc &
```

Which causes all future references to /etc to be interpreted as /helios/etc. This feature is particularly useful for mapping the Helios directory structure onto that of another operating system. Makefiles and shell scripts can then be ported with little modification.

Batch Server

The Batch Server provides a program scheduling service. It uses a job description language to define the program which is to be executed remotely. It allows the user to specify the time at which the program will be executed, its environment, and the interval at which it should be rescheduled (if required).

The syntax of the job description language is given here in BNF.

```
<batch_description>  ::= <job_description>  <batch_description>

<job_description>  ::= Tfname [arguments] '(' <parameters> ')'

<parameters>       ::=      'START'     <start_time>     <parameters>
                          | 'REPEAT'    <repeat_delay>   <parameters>
                          | 'PARENT'    <name>           <parameters>
                          | 'STATUS'    <status>         <parameters>
                          | 'PRIORITY'  <priority>       <parameters>
                          | 'SYSTEM'                     <parameters>
                          | 'ENVIRON'   <envv>;          <parameters>
                          | 'OBJECTS'   <objv>;          <parameters>
                          | 'STREAMS'   <strv>;          <parameters>
```

Explanation

<start_time> ::= [day]':'[month]':'[year]':'[hour]':'[minutes]':'[seconds]

The START parameter gives the time at which the job (Task Force) should be started. If no start time is given (or 0) then the Task Force is started immediately. The format of the time is given as

```
dd:mm:yyyy:hh:mm:ss
```

where dd is the day of the month, mm is the month of the year and yyyy is the year. The time is given in 24 hour format by hh:mm:ss. For example,

```
26:01:1989:10:00:00
```

specifies a time of 10am on the 26th January 1989.

If any of the time fields are replaced by a tilde ('~') then the start time given is added to the current time. Thus ~:~:~:2:20:00 will start the job 2 hours 20 minutes from the job submit time. If any date field is substituted by '~' then the current value is assumed. So if the current date is 26:01:1989:10:00:00, then a start time of 28:~:~:~:~:~ will run the job on the 28th Jan. at 10:00

<repeat_delay> ::= [hour]':'[minutes]':'[seconds]

The REPEAT parameter specifies the delay between consecutive instances of the job. If no repeat delay is given the Task Force is only executed once.

<status>

The STATUS field gives the job status. This can be 0 (the default) meaning the job is mortal and will be deleted on error. The value can also be 1 meaning that the job is immortal and will be automatically rescheduled on error; a value of 2 means the job will be deleted immediately.

<priority> ::= number

The PRIORITY field is defined to allow priorities at some future date. It is not currently used.

The SYSTEM parameter defines that the job is a system service and no environment will be sent to the Task Force when it starts executing. Unless this keyword is given, the job is a user job and is always sent an environment.

<envv> ::= name <envv>

The ENVIRON keyword is used to specify an environment string to the task force.

<objv> ::= object_name <objv>

The OBJECTS keyword is used to specify objects, which must exist, to the task force to be run. These are also passed in the environment.

<strv> ::= stream_name <strv>

The STREAMS keyword is used to specify open streams which are to be passed as the environment for a task force. Note that most C programs will normally require *stdin*, *stdout* and *stderr* to be defined within their environment.

TFname can clearly be the name of any executable Task Force (either an executable object or a CDL object), however the full context of the Task Force object must be given. An example script is given to illustrate the job description language.

```
/helios/lib/fastfiler -b -k/cache/bin (
                start  10:9:1990:14:20:0
                status 1
                streams /null /null /helios/error/filelog;
)


/helios/bin/garbage_collect (
                repeat 1:0:0
                status  0
```

```
                environ  NOCACHE DEADBLOCK;
                system
)
```

Having created the job description file, it can be submitted by using the system utility, **runb**.

```
runb testscript.jb
```

This creates the job and then passes it to the batch server for the component programs to be executed as required. If the job description is syntactically incorrect, **runb** will return with an error.

Fault Library

The Fault library is used to search a fault database for matching fault and error codes. There are two ways of using it: via *Fault()*, which searches the standard fault database in "/helios/etc/faults", or via the routines *fdbopen(), fdbrewind(), fdbfind()* and *fdbclose()*. Templates for all these functions are to be found in fault.h.

Fault

The first argument to this function is an error code. If the value is less than zero it is interpreted as a Helios error code. If it is greater than zero but less than or equal to the highest Posix error code it is treated as a Posix error code, otherwise it is treated as a Helios function code. The second and third arguments of Fault are a message buffer and its size. The message is added into this buffer as a null terminated string, and will be truncated if it does not fit.

Low Level Routines

These procedures allow a private fault database to be used to generate messages. The function *fdbopen* attempts to open the named file as a fault database and will return NULL on error; *fdbclose* closes a fault database. As the fault database is only searched forwards, *fdbrewind* repositions the search point at the start of the file.

The function *fdbfind* searches the fault database for an entry. The second parameter is the name of the fault class to be searched. The third parameter contains the code to be searched for; only the bits of this value described in the class entry's mask field will be compared. The remaining two parameters describe a buffer, the message corresponding to the code will be concatenated onto the end of the existing buffer contents, but only if the remaining space in the buffer is large enough.

Fault Database Format

A fault database is an ASCII file organised in lines. Any line beginning with a '#' is ignored. Numerical values may be given in decimal or in hex, preceded by "0x". A line beginning with a '!' is a class description: the first field is the class name, the second a mask indicating the bits that class occupies and the third optional field gives the C header prefix. A class ends with a line containing just "!!".

Within a class each line consists of an code name, a code value, and an optional message string. If the message string is not present, the code name is used.

Technical Changes

The V1.1 kernel is faster than previous versions, particularly when passing messages through a processor from one link to another. This kernel also contains support for attaching routines to the Event line. Other kernel features include monitoring of processor performance and a port table garbage collector.

The format of function codes has changed slightly to include a Retry field which is used by the Processor Manager to maintain a confidence level associated with each name in the name table. When this level drops below a certain threshold the name is removed from the name table, forcing a new distributed search for it. This provides an extra level of recovery in the face of processor and link crashes.

Module init routines are now called twice, once with a second argument of 0 and once with a second argument of 1 (this argument used to be undefined).

The arbitrary limit of 20 open files has been lifted in the Posix library; any number are now allowed. However the C library limit remains, largely due to compatibility issues involved in the implementation.

The new pipe-server provides bi-directional communication between processes. A new server "/pipe" is used to support this and should be used in preference to "/fifo" for inter-task communication. The shell and TFM now use it by default. Pipes, unlike fifos, have no internal buffering; communicating processes which are connected via the pipe-server interact directly, the only buffering available is that supplied by the runtime system.

When booted, the kernel now ensures that it synchronises internally, and with its parent, before continuing. For this reason BootLink() has been moved to the System Library, so any programs which call it should at the least be re-linked.

The C compiler has had a lot of optimisation work done on it, in particular branch chains are now eliminated along with unreachable code, and some peep-hole optimisation is performed at code generation. The head label of all loops is now aligned to a word boundary to make maximum use of the instruction fetch. All instances of mul have been converted to prod, since Helios makes no use of the error flag.

The C compiler was using the wrong rounding modes for the floating point conversion routines; this has now been fixed. The T800 floating point libraries are now correctly re-entrant, and the full set of rounding modes supported.

Open streams passed to a program in its environment (stdin, stdout, etc.) are now not actually opened until first used. This makes the startup of programs marginally faster.

Component Distribution Language

This section describes the enhancements that have been made to CDL for Helios 1.1. These changes are designed to simplify the description of complex, multi-component task forces. Compatibility with Helios 1.0 has been maintained as much as possible, the only change which may cause problems is the introduction of a precedence for parallel constructors.

Component Attribute 'code'

An additional attribute, 'code', is supported inside component declarations. Code is used to introduce a filename which specifies the actual piece of code to which the component refers. The effect of this is to remove the direct association between the name of a component and the code that it executes; for example, several components can now be defined, each having different resource requirements and the same code. For compatibility with Helios 1.0, if no 'code' attribute is specified, the name of the code defaults to the name of the component. For example, the following CDL script:

```
component musthaveT414
{
  code myprog;
  processor T414;
  memory 100000;
}

component musthaveT800
{
  code myprog;
  processor T800;
}
musthaveT414 ^^ musthaveT800 ^^ myprog
```

will execute three copies of 'myprog' in parallel. The first must execute on a T414 with at least 100000 bytes of memory, the second on a T800, and the third has no preference. Each component refers to the same code, which the CDL compiler locates by using the environment variable, PATH. Note that memory requirements must be specified in decimal.

Subscripted Component Declarations

Component declaration names can now be followed by one or more subscript names. Each subscript name is separated from the next by a comma, and the entire list is enclosed in square brackets. Within the body of the declaration, any stream name can include a list of subscript expressions which reference these subscript names. For example, the following component declaration:

```
component filter[i]
{
  code cat;
  streams <| pipe(i), >| pipe(i+1);
}
```

uses the subscript, i. This name is used subsequently in two subscript expressions within the body of the declaration; each expression is evaluated when the component is referenced. Whenever a component declaration is referenced within a task force definition, subscript values can be provided which are bound to the subscript names within the declaration. The aim of this is to be able to declare a component which, depending on subscript values it is referenced with, communicates on different streams. Many structures, such as arrays, can be defined by declaring a single component which has subscripted stream names. The structure is then defined by referencing this component with different subscript values.

Command names used with task force definitions may now be followed by a list of subscript values that are enclosed in braces and separated by commas. These subscripts may also be expressions, but for the purposes of this explanation they are assumed to be values. For example, the task force definition:

```
filter{0} ^^ filter{1} ^^ filter{2}
```

defines a task force which comprises three components. Each component refers to the component declaration for 'filter' (given above), but in each case different subscript values are passed into the declaration. The important point is that stream names with different subscript values refer to different streams. Thus 'filter{0}' reads from the stream 'pipe{0}' and writes to the stream 'pipe{1}', 'filter{1}' reads from stream 'pipe{1}' and writes to stream 'pipe{2}', and 'filter{2}' reads from stream 'pipe{2}' and writes to stream 'pipe{3}'. As it stands the task force definition is incomplete because the stream 'pipe{0}' has no component writing to it and the stream 'pipe{3}' has no component reading from it. For a task force to be complete, each stream that it uses must have one component reading from it and one writing to it. A complete task force definition might be as follows:

```
cat >|pipe{0} ^^ cat <| pipe{2} ^^
filter{0} ^^ filter{1} ^^ filter{2}
```

which is a pipeline made up of five cat commands.

Replicators

In the previous version of CDL, replicators preceded constructors. For example,

```
ls [3]| cat
```

is equivalent to

```
ls | cat | cat | cat
```

thus replicating the constructor and its preceding command three times. This is still supported, but Helios 1.1 supports a new variation in which the replicator directly follows the constructor. This form of replication does not define the constructor used to communicate with the preceding construction. For example,

```
|[3] cat
```

is equivalent to

```
cat | cat | cat
```

The first example in this section can therefore be written as:

```
ls | (|[2] cat)
```

This also affects the use of the interleave constructor (previously called the farm constructor). Our usual method of defining a farm construct,

```
control [3]||| worker
```

is still valid and is equivalent to:

```
control <> lb 3 (<> worker, <> worker, <> worker)
```

The load balancer command, lb, is inserted automatically by the CDL compiler and is given the job of interleaving the input and output from replications of 'worker'. The particular implementation of lb supplied in the standard Helios 'bin' directory attempts to balance the workload on each of the workers by way of a packet protocol; this can be overloaded as described in section 7.5 of the Helios Manual. We are not concerned here with the implementation details of lb, in fact we can ignore them altogether and just think of the input and output of the worker's as being interleaved.

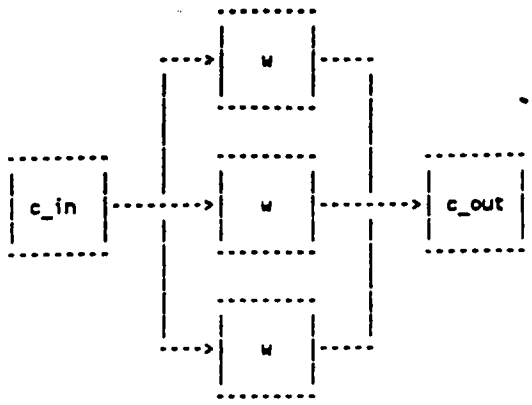
The interleave constructor can also be used with the new syntax for replicators. Using the new syntax, we would write the previous farm construct as:

```
control <> (|||[3] worker)
```

This is more flexible because we can now make use of an interleave construct as part of a pipeline. For instance,

```
control_in | (|||[3] worker) | control_out
```

defines a structure which looks like:



Note: Currently, the interleave constructor cannot be used as a binary operator but must always be used in conjunction with a replicator.

Multi-Dimensioned Replication

In Helios 1.0 only a single count was allowed within replicators. This has been extended in Helios 1.1 to a list of counts separated by commas, interpreted as multi-dimensioned replication. For example,

```
^^[2,3] node
```

is equivalent to

```
node ^^ node ^^ node ^^ node ^^ node ^^ node
```

The use of multi-dimensioned replicators will become apparent in the next section.

Note: Multi-dimensioned replication of pipes and subordinates does not produce a multi-dimensioned structure, but rather a one-dimensional pipeline or bi-directional pipeline just as if a single-dimensioned replicator was used. This may change in a future version and for this reason we recommend that multi-dimensioned replicators are only used with the simple parallel and interleave constructors. This should not present a problem as all structures should be achievable by use of a full component declaration.

Iteration Names

An iteration name may be associated with each dimension of a replicator. The replicated construction may contain command name subscript expressions and stream name subscript expressions involving these iteration names. When the replication is expanded, each successive replication is passed values for the iteration name from 0 to one less than the number of replications. The syntax for introducing iteration names is to precede the replication limit by the name followed by a '<' symbol. For example,

```
|[i<3] filter(i)
```

is equivalent to the task force

```
filter(0) | filter(1) | filter(2)
```

Thus our earlier example can now be written:

```
cat >| pipe(0) ^^ (|[i<3] filter(i)) ^^ cat <| pipe(2)
```

The scope of each iteration name is the entire construction being replicated. Since these constructions may contain further replication this scope may contain 'holes' where the same iteration name is redefined. A reference to an iteration name obtains the value of the most closely nested one of that name. The scope rules are similar to the scope rules of variables in a block-structured language. For example,

```
^^[i<2,j<2] (a(i,j) ^^ |[i<3] b(i,j))
```

is equivalent to

```
(a(0,0) ^^ b(0,0) | b(1,0) | b(2,0)) ^^
(a(0,1) ^^ b(0,1) | b(1,1) | b(2,1)) ^^
(a(1,0) ^^ b(0,0) | b(1,0) | b(2,0)) ^^
(a(1,1) ^^ b(0,1) | b(1,1) | b(2,1))
```

The 'i' and 'j' used with the 'a' component refer to the iteration names defined in the first replicator, as does the 'j' used with the 'b' component. The 'i' used with the 'b' refers to the 'i' defined in the second replicator.

The value of an expression involving iteration names can be passed as an argument to a component. The expression is placed in the list of arguments to the component in the task force definition and is distinguished from a normal argument by preceding it by a '%' character. For example,

```
^^[i<3] node(i) %i+1
```

is equivalent to

```
node(0) 1 ^^ node(1) 2 ^^ node(2) 3
```

Note: Whereas stream name subscript expressions are used to form complete stream names, command name subscript expressions, once used to expand the referenced component declaration, are of no further significance.

Precedence of Constructors

In Helios 1.1 each constructor has a unique precedence, as opposed to the common precedence of all constructors in Helios 1.0. The constructors have the following order of precedence, in ascending order from left to right:

```
^^  |||  |  <>
```

For example,

```
a <> b | c
```

is equivalent to

```
(a <> b) | c
```

rather than

```
a <> (b | c)
```

(note the use of parentheses to override the precedence) and

```
a <> b | c ^^ d | e
```

is equivalent to

```
((a <> b) | c) ^^ (d | e)
```

This affects the way in which streams are allocated and the configuration of the resulting task force.

Automatic Allocation of Streams

The way in which streams are automatically allocated for task forces can be encompassed by the rules outlined below.

A simple parallel constructor, ^^, defines no communication between its operands.

A pipe constructor, |, defines a single communication between its operands. In general, for the task force,

```
A | B
```

file descriptor 1 of A is connected to the file descriptor 0 of B. Note that in this and in subsequent examples A and B may themselves be task forces.

A subordinate constructor, < >, defines a pair of communications between its operands. For example, in

```
A <> B
```

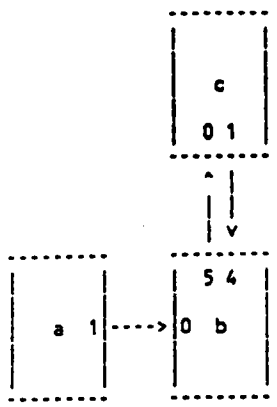
file descriptor 1 of B is connected to the first auxiliary input of A (this is defined to correspond to file descriptor 4 at the POSIX level), and the first auxiliary output of A (POSIX file descriptor 5) is connected to the file descriptor 0 of B.

The order of allocation of streams for a task force is defined by the precedence of the constructors. Thus for,

```
a | b <> c
```

first the streams for the subordinate constructor are allocated and then the streams for the pipe constructor. Streams for constructors of the same precedence are allocated left to right.

Once a file descriptor of a component has been overloaded it becomes a hidden internal stream and cannot subsequently be overloaded. So, for the previous example, file descriptor 0 of component 'c' is initially overloaded by the allocation of streams for the subordinate constructor and consequently is not further overloaded by allocation of streams for the pipe constructor. The example produces a structure that looks like:



where each box represents a component; a line connecting two boxes represents a stream, with the arrow giving its direction, and number representing the file descriptor.

The allocation of streams within an auxiliary list is a special case. This is because each of the constructors within the list has the same left hand operand, as in the following example.

```
master (<> slave1, | slave2, <> slave3)
```

In this case, both subordinate constructors and the pipe constructor have the same left operand, master. Each such constructor in an auxiliary list uses successive auxiliary streams of its common component, with the proviso that they are allocated in pairs. Remember the first auxiliary stream corresponds to POSIX file descriptor 4. An input is allocated on an even numbered file descriptor and an output on an odd numbered file descriptor. So, in this example, 'master' communicates with 'slave1' on file descriptors 4 and 5, 'slave2' on file descriptor 7, and 'slave3' on 8 and 9.

Subscript Expressions

Subscript expressions are written in standard arithmetic format and may contain integers, subscript names, binary operands, unary operands, and parenthesis. The binary operands supported are as follows:

- \+ addition
- \- subtraction
- * multiplication
- % remainder

+ and - are also supported as unary operands.

CDL Compiler

In addition to supporting the enhancements described in this document, the latest version of the CDL compiler has a few new features which are outlined here.

A new option, **-e**, is supported. This option causes the code for each component of the task force to be included in the resulting CDL object file. This is useful for producing CDL object files for transporting to other systems.

As mentioned earlier, the latest compiler validates the use of streams. Every stream used by a task force must have one reader and one writer and any other combination will result in a compilation error.

Examples

This section contains some example CDL scripts. In each we are not interested in the details of the application code but merely in how to generate the correct structure in terms of components and connecting streams.

Example 1

Here we wish to define a task force consisting of 10 components, where each component is a filter, reading on file descriptor 0 and writing on file descriptor 1. We require these streams to be connected to form a 'ring' of components. To achieve this we first declare the following component:

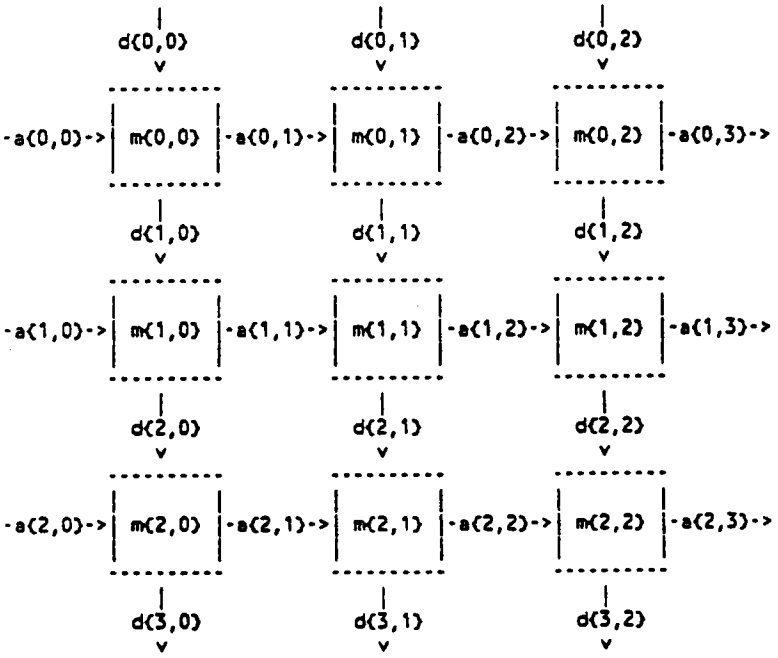
```
component node[i]
{
  streams <| pipe(i), >| pipe((i+1)%10);
}
```

The actual task force definition simply consists of a replicated structure of 10 invocations of 'node' running in parallel.

```
^^[i<10] node(i)
```

Example 2

For our second example we wish to generate a two-dimensional matrix of components, where each component has two inputs and two outputs. A diagram is the easiest way to show what we require.



Sensible subscript values for components and stream names is vital in defining a complex task force; it enables the stream name subscripts of an expression to be expressed in terms of the subscript values of the component. This is exactly what we have done in the following component declaration:

```
component mult[i,j]
{
  streams <| across(i,j), >| across(i,j+1),
          <| down(i,j),   >| down(i+1,j);
}
```

Here 'mult' refers to the 'm' in our diagram, 'across' to 'a', and 'down' to 'd'. The task force definition replicates this component, passing the required subscript values. This looks like:

```
^^[i<3,j<3] mult(i,j)
```

Note: This example does not define a complete task force.

New Syntax

This section gives an updated formal definition of the CDL syntax.

```
<script>       ::= ( <declaration> ) <taskforce>

<declaration>  ::= 'component' <name> <sub-names>
                   '(' ( <attribute> [';'] ) ')'

<attribute>    ::= 'code' <name>
                 | 'processor' <ptype>
                 | 'puid' <name>
                 | 'attrib' <attriblist>
                 | 'memory' <size>
                 | 'life' <life>
                 | 'time' <number>
                 | 'priority' <number>
                 | 'streams' <streamlist>

<ptype>        ::= 'T414' | 'T800' | 'ANY'

<attriblist>   ::= <attrib> ( [','] <attrib> )

<attrib>       ::= <name> [ '[' <number> ']' ]

<life>         ::= 'mortal' | 'immortal'

<streamlist>   ::= <stream> ( [','] <stream> )

<stream>       ::= <mode> <name> [ <sub-exprs> ]

<mode>         ::= '<' | '>' | '>>' | '<>' | '>|' | '<|'

<taskforce>    ::= <interleave> ( '^^' <interleave> )
                 | '^^' <replicator> <interleave>
                 | <replicator> <construction>

<interleave>   ::= <pipeline>
                 | '|||' <replicator> <pipeline>
                 | <replicator> <construction>

<pipeline>     ::= <subordinate> ( '|' <subordinate> )
                 | '|' <replicator> <subordinate>
                 | <replicator> <construction>

<subordinate>  ::= <command> ( '<>' <command> )
                 | '<>' <replicator> <command>
                 | <replicator> <construction>

<command>      ::= <simple cmd> [ <auxlist> ]
                 | '(' <taskforce> ')'  [ <auxlist> ]

<construction> ::= '^^' <interleave>
                 | '|||' <pipeline>
                 | '|' <subordinate>
                 | '<>' <command>

<auxlist>      ::= '(' <aux> ( ',' <aux> ) ')'

<aux>          ::= '|'  <taskforce>
                 | '|<' <taskforce>
                 | '<>' <taskforce>

<simple cmd>   ::= <name> [ <sub-exprs> ]
                   ( ( <arg> | <stream> ) )

<mode>         ::= '<' | '>' | '>>' | '<>' | '<|' | '>|'
```

```
<sub-names>     ::= [ <name> , { <name> ) ]

<sub-exprs>     ::= '(' <sub-expr> { <sub-expr> ) ')'

<sub-expr>      ::= <unary-expr> { <binary-op> <sub-expr> )

<unary-expr>    ::= <number>
                  | <name>
                  | <unary-op> <sub-expr>

<binary-op>     ::= '*' | '%' | '+' | '-'

<unary-op>      ::= '+' | '-'

<replicator>    ::= '[' <dimension> { , <dimension> ) ']'

<dimension>     ::= [ <name> '<' ] <number>

<name>          ::= sequence of printable characters

<number>        ::= sequence of decimal digits
```

New Commands in Helios 1.1

This section provides a detailed description of the commands: **boot**, **c**, **cache**, **dlink**, **elink**, **fg**, **kill**, **lbpcat**, **lcontrol**, **map**, **netversn**, **reset**, **run**, **runb**, **startns**, and **tcp**.

boot

Purpose: Boot the given subnetwork.

Format: *boot <subnet_name>*

Description:

The subnetwork (this may just be a single processor) will first be reset, and then the network server will attempt to boot it. The command will not return until the whole subnetwork is booted.

```
boot /net/clustera
```

Note: that there is a 60 second timeout on this command, so when booting very large subnetworks this command may return with a timeout error, although the subnetwork will still be booted.

c

Purpose: Compiler driver.

Format: *c [opts] <filename> [<filename> ...]*

Description:

This command is used to compile and link a program. It takes a list of files and decides what to do with them according to their suffix. The file names supported are

.a	Macro assembly language (AMPP source)
.c	C language
.f	FORTRAN language
.o	Object file format
.s	Assembly language

If no other arguments are given, the program compiles the programs for the languages specified as .c or .f, assembles any .s files and then links all the resulting binaries along with any supplied .o files into an executable program called a.out. A large number of options can be used to alter the behaviour of the program as follows.

-b	Don't link with standard libraries (fplib and fpclib).
-c	Compile/Assemble only, don't link.
-d<name>	Specify output file name for library .def compilations.
-e[6|7]	Enforce Fortran standard.
-h<val>	Specify heap size of program.
-j	Join objects but don't link.
-l<name>	Link with standard library <name> (/helios/lib/<name>lib.def).
-m	Compile code for libraries.
-n	Don't actually execute commands (implies -v).
-n<string>	Specify object name of program.
-o <name>	Specify output name (default *.o or "a.out")
-p	Compile code for profiling.
-q<string>	Enable compiler debugging features.
-s<val>	Specify stack size of program.
-t	Compile code for tracing.
-v	Verify command being executed.
-w[acdfpsvz]	Suppress warnings.
-A<flag>	Pass <flag> direct to linker.
-B	Do not link with any libraries. Do not perform objed.
-C	Perform array bound checking (F77).
-D<name>	#define <name> (C)
-D<name> = <val>	#define <name> to be <val> (default <val> is 1) (C).
-F[fghmsv]	Enable compiler features ('s' turns off stack checking & 'g' suppresses insertion of function names in code) (C).
-I<dir>	Specify a directory to be searched for #include files.
-L<name>	Link with standard library <name> (/helios/lib/<name>.def).
-M<name>	Produce map file <name> (F77).
-O	Optimise code, perform full link.
-S	Produce textual assembler output in *.s, don't link.
-V	Pass on verbose flag to executed commands.
-T[4|8]	Specify Transputer type.
-W<val>	Specify warning level (F77).
-X<val>	Specify cross reference width (F77).
-help	This message.

cache

Purpose: To load and retain a program module in memory.

Format: *cache [-t | -m <machine>] program*

Description:

A new option, -t, has been defined for this command. In normal use, the program is loaded into the local loader, and the use of cache prevents it from being unloaded when it finishes. When using the TFM, it is not desirable to cache the program in a local loader as it may be executed on another processor, and this will result in the program being reloaded from disc because it is not in that processor's own loader.

The -t option has been added so that the command is cached in the TFM. This means that the program is cached in a processor which will always be used for executing that command. This eliminates the need for reloading. You should note that this command should always be followed by **rehash**, and that */tfm* should be included in the wordlist for the environment variable, PATH.

dlink

Purpose: Disable the given link.

Format: *dlink <subnet_name> <link_number>*

Description:

Disables the given link. This will set the given link into dumb mode.

```
dlink /net/clustera/04 1
```

elink

Purpose: Enable the given link.

Format: *elink <subnet_name> <link_number>*

Description:

Enables the given link. This will set the given link into intelligent mode and attempt to enable it.

```
elink /net/clustera/04 1
```

fg

Purpose: To bring a job to the foreground.

Format: *fg [<job_name>]*

Description:

This shell command brings the specified job to the foreground. If no argument is supplied, the current job is assumed. The following character sequences can be supplied as arguments, and have the meanings shown:

Job Name	Description
%<job_number>	The job with the specified job number.
%%	The current job.
%+	The current job.
%-	The previous job.

kill

Purpose: To terminate the specified job.

Format: *kill <job_name> | <processid>*

Description:

This shell command is used to terminate the specified job; the job can be identified by either its job name or its process identification number. The following character sequences can be supplied as arguments, and have the meanings shown:

Job Name	Description
%<job_number>	The job with the specified job number.
%%	The current job.
%+	The current job.
%-	The previous job.

lbpcat

Purpose: Echo load balancer packets.

Format: *<component> [n]||| lbpcat*

Description:

This is a simple utility which just echo's load balancer packets. It can be used as a test worker-task which simply echo's all packets it is sent; for example:

```
controller [20]||| lbpcat
```

tests that the controller and load balancer are creating and routing packets correctly.

lcontrol

Purpose: Convert stream to line mode.

Format: *<component> lcontrol < > lb ....*

Description:

The lcontrol utility is a simple front end for the Helios load balancer. As described in *The Helios Operating System* (7.5), the load balancer uses a specific packet protocol for all communication with a control process. It is however possible to change the protocol of the streams which connect the load balancer to the worker tasks into line mode, so that standard text processing utilities can be used as worker processes. This demands that a utility be provided to convert a stream in line mode to a packet stream for input to the load balancer; this functionality is provided by lcontrol. So to create a distributed concatenator:

```
cat <filename> | lcontrol [4]||| mycat
```

In this case lcontrol is used only to set the load balancer into line mode and to pass on the lines output by the controlling 'cat'.

map

Purpose: Display activity in Helios node.

Format: *map*

Description:

When the command is started it displays a help page listing valid commands and a summary of the map format. Each command is selected by a single key press, as shown below

q Q ESC	Terminate 'map'.
h H	Display help page.
-	Halve sample rate.
+	Double sample rate.
any other key	Resize display and redraw.

The map consists of four fields. The top row displays the total amount of memory which has been allocated, the amount which is free, and the number of bytes which each character in the map represents. Along the right hand side of the display is a table showing a list of active tasks and a letter which has been assigned to that task; these letters are then used in the main map to represent each of the tasks. The main map occupies the centre of the screen; this shows the allocation of system heap. Each character in the map represents a number of bytes which is specified in the top row of the display. Character positions which are occupied by '.' represent free memory, '#' represents memory which has been allocated to the system, and '@' and '?' are used to represent unidentified allocations. All letters which appear in the map show the amount of memory which has been allocated to a task, and digits represent shared libraries. At the bottom of the display is a graph showing the current processor load; the horizontal bar at the end of the graph marks the maximum load, and '=' shows the current loading.

netversn

Purpose: Licencing details for network server.

Format: *netversn <subnet_name>*

Description:

This command gives licencing information about the network server. It displays the information about the type of licence which was issued (Single Machine or Network), the distributors identification code, and the server's serial number.

reset

Purpose: Reset the given subnetwork.

Format: *reset <subnet_name>*

Description:

Reset the given subnetwork. The main restriction to the application of this command is that it must not be used to reset the whole subnetwork, as this would destroy the root network server and put the network into an unrecoverable state. If you wish to reset the whole network you should re-boot Helios.

run

Purpose: Run a command in its own window.

Format: *run <command>*

Description:

This command creates a window, executes the specified command, and closes the window when the command terminates. The command is located by using the environment variable, path.

runb

Purpose: Submit job description to batch server.

Format: *runb <file>*

Description:

This command submits the specified job description file to the batch server, and then returns to the caller. A full description of the file's syntax is given elsewhere in this note; see *The Batch Server*.

startns

Purpose: Start local network server.

Format: *startns [options] <subnet_name>*

Description:

This is used to startup a local network server. This command can be entered into the shell, or used in the startup file, initrc. The options are passed to the network server and are as shown below:

- -r Reset everything
- -nr No reset; do not attempt to automatically reset on booting.
- -nt Do not create task force manager for this network.
- -nb Do not boot this network.

The above options may also be combined; for example, *-nmt* specifies that the network is not to be reset and that no task force manager is to be created (a task force manager is created by default).

tcp

Purpose: Copy files, converting CR/LF to LF.

Format: *tcp <filename> <filename>*
tcp <filename> [<filename>] ... <dir>

Description:

This copies one or more text files, much in the same way as **cp** but translating CR/LF to LF. This is required when copying text files from an external filing system such as MS-DOS, which stores end of line as CR/LF, into an internal filing system such as the Helios filing system or ram disc which uses just LF. The reader is also advised to consult the Helios manual for details of **xlatecr**.

Network Support

This sections describes how Helios 1.1 can be used for networking. It should be noted that the networking extensions to Helios are not available in all upgrades. If in doubt, you should contact your distributor.

In order to run networked Helios you will need to have two different copies of Helios booted from two separate hosts. Each host should be responsible for one subnetwork; at the minimum this means one transputer each. The resource maps used in both machines must correspond to the subnetwork to be booted by the network server in that subnetwork and must be given a unique subnet name. Any program requested to be run by the TFM within either subnetwork will not use any processor in the other subnetwork.

Each subnet description must describe any *external links*; these are the links which are used to cross connect the two subnetworks. External links are identified by number, normally starting at 0. The syntax used is that instead of specifying a terminal component name such as ~01 you must specify ext[n] where n is the external link number. Each end of the external link must be defined, one end in each subnetwork. An example should help clarify this.

```
subnet /ClusterA (
        CONTROL Rst_Anl [/ClusterA/00];
                terminal 00 ( ~IO, ext[0]  ,  ,;
                                        HELIOS;
                                        Mnode   Rst_Anl [im_ra_b4.d];
                                        ptype T414; )
                terminal IO ( ; IO;)
)
```

Here the first subnet ClusterA has one transputer. Link 1 is designated the initial external link. This resource map is compiled and a network server started in order to boot this and run the associated TFM. In this case, the network server should be started by **startns**; for example,

```
startns /helios/etc/ClusterA.map
```

The following example defines the map for ClusterB, which shows the external link connected to link 3 of the single transputer in this subnet. The other copy of Helios should be booted and run with this map provided for the network server and TFM. Once again, the network server should be started by **startns**.

```
subnet /ClusterB (
        CONTROL Rst_Anl [/ClusterB/00];
                terminal 00 ( ~IO,  ,  , ext[0];
                                        HELIOS;
                                        Mnode   Rst_Anl [im_ra_b4.d];
                                        ptype T414; )
                terminal IO ( ; IO;)
)
```

In order to cause the network to become connected, a further version of the network server must be run in one transputer somewhere on the network. This must be started by **startnet**; the optional flags, *-nt*, should be used if users do not wish to share processors. This netwide network server must be provided with a network map which identifies the ways in which the subnetworks are interconnected via the external links. In our simple example the map to be used would look as follows.

```
subnet /Net (
                subnet ClusterA ( /Net/ClusterB; HELIOS; )
                subnet ClusterB ( /Net/ClusterA; HELIOS; )
)
```

The initial external link is used as the first item in the external connection list. If there were any further external links they would follow the first, separated by commas. The subnetwork addresses in the link definitions must include the full pathname of the target subnetwork; '~' is not valid at this level.

Network Commands

This section describes the set of Network Control commands distributed with this toolkit. These commands are exclusive to the networking Helios system and provide a command line interface to the Network Control system.

Control System

The network control system is provided by a distributed Helios server called the Network Server (NS). This service is responsible for booting the network and for its subsequent control. The network must first be defined by the use of a text file called a resource map; the format of which is described in the Appendix. The resource map is read by the NS at system boot time and is used to boot the defined network. Once the network is booted the NS automatically installs the Task Force Manager hierarchy which enables the automatic allocation of programs (task forces) to processors.

The network control commands provide a simple command line interface to a resident library called the Network Control Library (net_ctrl). This is a library of routines which send requests direct to the relevant NS to perform the control functions. For example, the processor, /net/machine1/04, can be reset by:

```
reset /net/machine1/04
```

Reset calls a function, Reset(), in net_ctrl which sends a reset request to the network server responsible for this processor (/net/machine1/ns). It is similarly possible to reset a whole subnetwork, for example:

```
reset /net/machine1
```

which will, if possible, reset all the processors in subnetwork /net/machine1.

It is important to note that in the current version of Helios these commands are potentially executable by any user. The effects of boot and reset are quite terminal, so it is desirable that access to these commands be restricted to privileged users; this will be implemented in future versions of the operating system.

Commands

clnames <*subnet_name*>

Clear all the name tables in the given subnetwork. Helios provides a distributed name service, with each processor maintaining its own table of object addresses (see "The Helios Operating System" Chapter 15.2). It is desirable to flush these name tables under certain circumstances in order to force a new search for a given object. Entries in the name table for objects local to the processor e.g. a local server are not removed.

```
clnames /net/clustera
clnames /net/clustera/04
```

connect <*subnet_A_name*> <*subnet_B_name*>

Connect the two subnetworks. It must be possible to access the NS responsible for <subnet_A_name> from the processor on which this request is issued. So the request

```
connect /net/clustera /net/clusterb
```

will not succeed if initiated from subnetwork /net/clusterb, but the following should work:

```
connect /net/clusterb /net/clustera
```

cupdate *<subnet_name>*

Update the network context of the given network. This command is used to update a partial network context to the full network context, and may be used when re-connecting errant subnetworks.

```
cupdate /clustera /net/clustera
```

Will update the network address (context) of every processor in the subnet clustera, and update the context of the required NS's and TFM's.

dconnect *<subnet_A_name>* *<subnet_B_name>*

Disconnect the two subnetworks. For example:

```
dconnect /net/clustera /net/clusterb
```

lrecon *<link0_mode>* *<link1_mode>* *<link2_mode>* *<link3_mode>*

<link_modes> = 1 or 2 (dumb or intelligent)

This utility does not use the network control library at all. It calls the kernel directly to change the state of the local processor links, using the kernel Reconfigure. It is useful for changing link modes from the command line.

```
lrecon 2112
```

Will disable links 1 and 2.

lstatus *<subnet_name>* *<link_number>*

Returns the status of the given network link. The status is returned in the format of the LinkConf structure (see link.h and config.h). The State and Mode are most useful as these define the current status of the link.

```
lstatus /net/clustera/04 0
```

Will return the status of this physical link.

native *<subnet_name>*

Revert the given subnetwork to the native state. This involves terminating the network control servers (NS & TFM) in this subnetwork and resetting the member processors. It is not possible to revert the whole network to native in this way (see boot). This will also disable any links which disconnect this subnetwork, i.e. any links which connect this subnetwork to the still active outside world.

```
native /net/clustera
```

Note this command should be used with care as it will kill any programs running in the target subnetwork. It does not currently terminate user programs.

sfnc *<subnet_name>* *<system_function>*

This command updates the system function entry in the distributed database of the NS and TFM. The system function defines what a subnetwork may be used for. Only processors with the function HELIOS will be considered for running user programs. If the user wishes to prohibit further placement of programs in a particular processor this command can be invoked with the SYSTEM function.

NATIVE = 1 (You should use the native command not this option)
HELIOS = 2 This is a HELIOS subnetwork; can load user programs
IO = 3 This is an IO Subnetwork
SYSTEM = 4 This is a SYSTEM subnetwork; cannot load user programs

For example,

```
sfnc /net/clustera/03 4
```

smemory *<subnet_name> <memory_size>*

```
<memory_size> = new memory size for the subnetwork in bytes.
```

This command updates the memory entry in the distributed database of the NS and TFM. The memory field defines the total amount of memory available in the given subnetwork, and is used in the mapping of task forces, where memory requirements have been specified in the CDL definition.

```
smemory /net/clustera 2000000
```

sptype *<subnet_name> <processor_type> <number_processors>*

<processor_type> = 2 or 3 or 4 (T414, T800 or 68000)
<number_processors> = number of processors of given type

This command updates the processor types entry in the distributed database of the NS and TFM. The processor types field defines the total number of processors of each type in the given subnetwork.

sstatus *<subnet_name>*

This returns the internal NS state for this subnetwork, this is really only of any use when trying to determine why a subnetwork has not been booted. Valid states include:

ACTIVE = 1 The subnetwork is active (fully booted)
PENDING = 2 The subnetwork is changing to ACTIVE (partially booted)
UNKNOWN = 8 State unknown (still trying to determine state)
UPDATE = 16 The NS is updating its database

startnet *[options] <map_name>*

This command is used to startup a netwide network server. It should be used after startns has been used to start up the local network servers. Valid options for the Network Server are as follows:

-r Reset everything.
-nr Do not attempt to automatically reset on booting (set by default)
-nt Do not create task force manager for this network.
-nb Do not boot this network.

The above arguments may also be combined; for example, *-nrnt* specifies that the network is not to be reset and that no task force manager is to be created.

By specifying the option, *nt*, users are allowed to share the available processors. In general this is inadvisable as there is currently nothing to stop a user from resetting another user's processor.

The Session Manager

The Session Manager (SM) is a system service responsible for creating user sessions and for restricting the total number of active sessions in a Helios system. It reads in the standard Helios password file, etc/passwd, and generates an internal database of users, /sm/userdata. The list of potential users can thus be read from the SM by opening the directory, /sm/userdata, and reading it. Although the SM does not currently support the reading of the userdata entries themselves, future versions will allow this.

The session manager services requests to create a session. When the SM is requested to create a session it expects to be given an initialised SessionInfo data structure which identifies the user. Having validated the user name and password, the SM creates a session entry and returns the created session; this represents a single users initialised session data. Several Open requests may then be made to create session environments, each one generating a RequestEnv request to produce the environment. The resulting environment which is created for each session includes the root program name (usually a shell), the users home directory, the users console and keyboard streams and other attributes. A record of each active session is maintained in the directory, /sm/users. A list of the users who are currently logged into the Helios network can then be obtained by listing the contents of this directory.

The interface to the SM is usually provided by a login worker process called **smlogin**. This is equivalent to the basic login program but it uses the SM to verify the username and password and to create the login shell environment as described.

The password file may be updated to change the parameters or number of valid users. In the present implementation it is not possible for users to change their passwords; to do this, the system administrator must change the password file directly.

Format of password file

```
UserName:Passwd:GID:UID:Comment:HomeDirectory:RootProgram [args]

GID = Group Id
UID = User Id
Comment = a text comment (usually full name of user)
HomeDirectory = the users home directory
RootProgram = the root program (usually /helios/bin/shell)
```

The utility, **smlogin** is a login service. It displays the login prompt, creates a session (within the session manager), and a login shell. When the login shell is exited, **smlogin** will re-display the login prompt to allow another user to login. **smlogin** will also support login through a remote console if it is provided with the stream which will be opened for the login console. For example, to login via a remote window server:

```
smlogin -d /fred/00/window/tty2 &
```

will create a login worker for the stream /fred/00/window/tty2.

New functions

The following functions have been added:

*word *InitProcess(word *stack, VoidFnPtr entry, VoidFnPtr exit, word *display, word nargs);*

Initialise a process for execution. *Stack* points to the top of the memory to be used as the stack, *entry* is the code to be executed in the process and *exit* the return address for when this returns. *Display* points to the initialised display passed to the initial call. *Nargs* is the number of bytes of arguments to be passed in. The stack is initialised according to the standard calling conventions and a pointer returned to the space left for the arguments.

*void StartProcess(word *p, word pri);*
A process initialised by InitProcess is started at the given priority.

void StopProcess(void);
Halt the current process.

*word GetPortInfo(Port port, PortInfo *info);*
The PortInfo structure provided is filled in with information about the port.

*void FreeMemStop(void *mem);*
The memory block is freed and the current process halted. This is used to allow the stack on which the current process is executing to be released without it being re-allocated before the process has a chance to stop itself.

*void SignalStop(Semaphore *sem);*
Signal the semaphore and halt the current process. Like FreeMemStop this is used to prevent problems in memory allocation.

word Configure(LinkConf newconf);
This is used to re-configure a single processor link. This function should be used instead of *Reconfigure()*.

*Stream *PseudoStream(Object *object, word mode);*
Manufacture a stream of Type_Pseudo to the given object. Unlike normal streams this will not be opened. It may be used wherever a normal stream can be used, and will be opened automatically if necessary.

Changed Functions

The following functions have changed in specification:

*PUBLIC word GetEnv(Port port, Environ *env);*
The Port argument has been added, to allow environments to be passed to any port.

Extended Functions

The specifications of the following functions have been extended:

*void *Malloc(word size);*
if size = -3 result is total size of heap.

Appendix A: Resource Map grammar

The formal semantics of the Resource Map are given here in the form of an extended BNF.

```
<Resource_Map>      -> <declaration> end

<declaration>       -> <subnets> <declaration>
                    |

<subnets>           -> SUBNET name '{' <def> <subnets> '}'
                    | TERMINAL name '{' <links> <fields> '}'

<def>               -> [<control>] [<links>] [fields]

<control>       -> CONTROL <type> <control>;

<links>             -> <addr>, <links>;
                    | ;

<addr>              -> address
                    | -name
                    | ext'['number']'

<fields>            -> <function>; <fields>
                    | PTYPE <processor_type>; <fields>
                    | MEMORY <memory>; <fields>
                    | ATTRIB <device_attributes>; <fields>
                    | MNODE <master_node>; <fields>

<function>          -> HELIOS
                    | IO
                    | NATIVE

<processor_type>    -> T800
                    | T414
                    | T212
                    | 68000
                    | 80286

<memory>            -> number (decimal)
                    | #number (hexadecimal)

<device_attributes> -> <attrib>, <device_attributes>
                    | <attrib>

<attrib>            -> name'['number']'
                    | name

<master_node>       -> <type>;

<type>              -> Rst_Anl '['<qualifier>']'
                    | CONFIGURE '['<qualifier>']'

<qualifier>         -> name
```

The principal syntax change between version 1.0 and version 1.1 is the change of delimiters used for the 'terminal' definitions. All 'subnet' and 'terminal' definitions are now surrounded by braces. The fields have been modified by the inclusion of reserved tokens to identify all the possible fields. So, for example, the processor type is now given as

```
ptype  T800;
```

instead of just

7800;

The only other major change has been the inclusion of an external link type 'ext[n]', but this is only applicable when connecting networks.